

Open in app ↗



Search

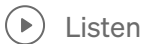


Avro



Suresh Podeti

5 min read · May 28, 2023



Listen



Share



More



Apache Avro official logo

Apache Avro is an open source **data serialisation system**. Used to encode data as **sequential bytes** before **storing** and **transporting** via network. This provides **space efficient encoding**, and allows **schema evolution** honouring forward and backward compatibility.

Avro uses:

1. While we need to **store the large set of data on disk**, we use Avro, since it helps to **conserve space**.
2. We get a better **remote data transfer throughput** using Avro for **data exchange services** like Kafka, Hadoop, and RPC etc..
3. Felicitates **backward and forward compatibility**.

Avro uses a **schema** to specify the structure of the data being encoded. It has two schema languages: one (**Avro IDL**) intended for **human editing**, and one (based on **JSON**) that is more easily machine-readable.

```
# Avro JSON based schema
{
  "type": "record",
  "name": "Person",
  "fields": [
    {"name": "userName", "type": "string"},
    {"name": "favouriteNumber", "type": ["null", "long"]},
    {"name": "interests", "type": {"type": "array", "items": "string"}}
  ]
}
```

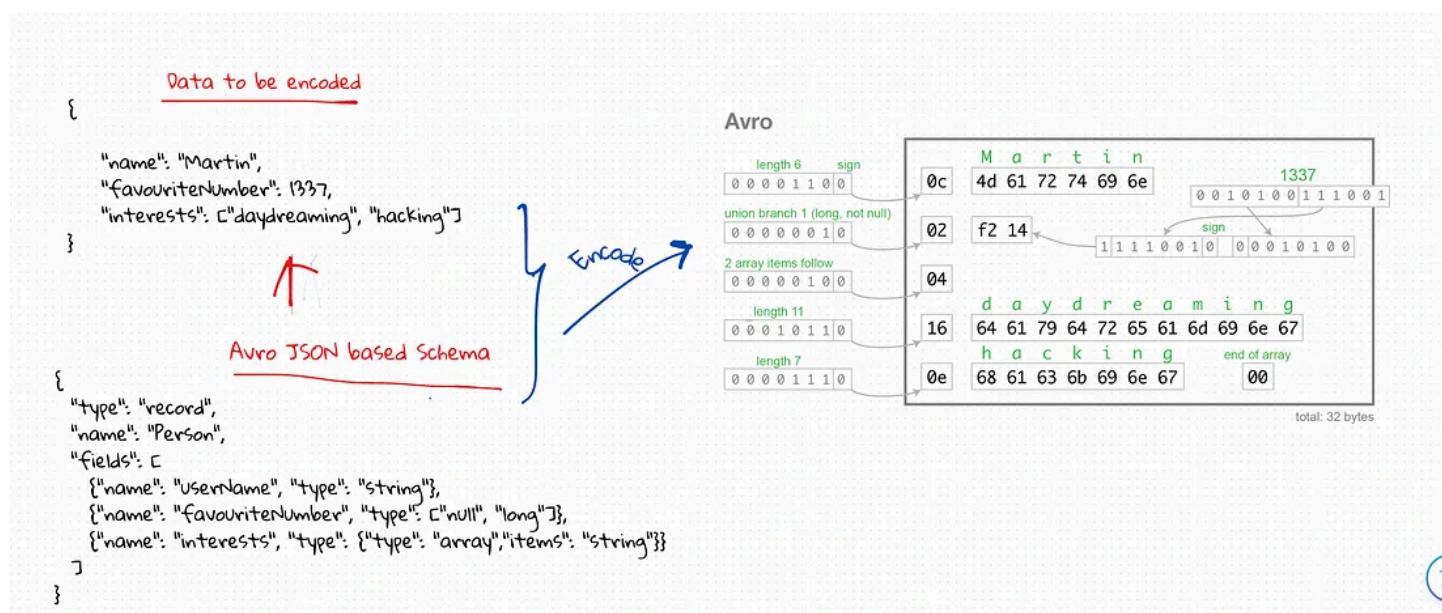


Fig 1.1: Data serialziation using Avro JSON schema

Schemas are composed of **primitive types** (null, boolean, int, long, float, double, bytes, and string) and **complex types** (record, enum, array, map, union, and fixed).

With Avro, when an application wants to **encode** some data (to **write it to a file or database, to send it over network, etc.**), it encodes the data using whatever version of the schema it knows about. This is known as the *writer's* schema.

When an application wants to **decode** some data (**read it from a file or database, receive it from the network, etc.**) , it is expecting the data to be in some schema,

which is known as the *reader's* schema.

The key idea with Avro is that the **writer's schema** and the **reader's schema** don't have to be the same—they only need to be **compatible**.

Schema evolution rules

With Avro, forward compatibility means that you can have a new version of the schema as writer and an old version of the schema as reader. Conversely, backward compatibility means that you can have a new version of the schema as reader and an old version as writer.

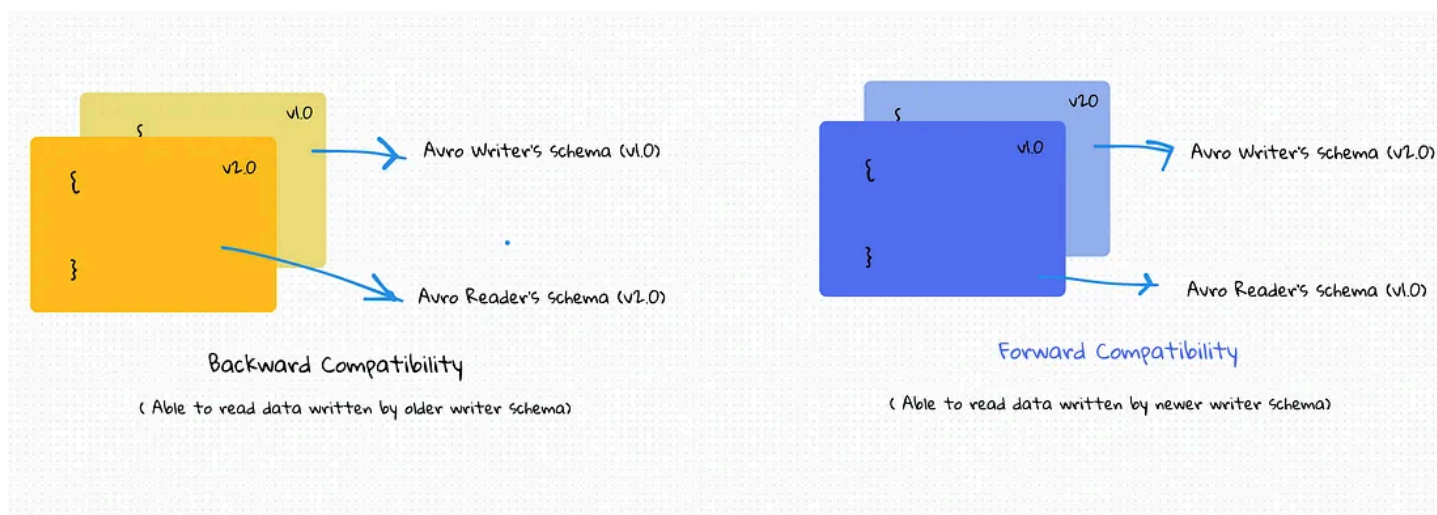


Fig 2.0: Avro backward and forward compatibility

So how does Avro support schema evolution?

Different order of readers and Writer's schema

Writer's schema for Person record

Datatype	Field name
string	userName
union {null, long}	favoriteNumber
array<string>	interests
string	photoURL

Reader's schema for Person record

Datatype	Field name
long	userID
union {null, int}	favoriteNumber
string	userName
array<string>	interests

It's no problem if the writer's schema and the reader's schema have their fields in a different order, because the **schema resolution matches up the fields by field name**. If the code reading the data encounters a field that appears in the writer's schema but not in the reader's schema, it is ignored. If the code reading the data expects some field, but the writer's schema does not contain a field of that name it is filled in with a default value declared in the reader's schema.

Adding a field with / without default value

Adding a new field with / without a default value		
	Backward Compatibility	Forward Compatibility
without	✗	
with	✓	

To maintain compatibility, you may only add **field that has a default value**. When a reader using the new schema reads a record written with the old schema, the **default value is filled in for the missing field**.

If we were to add a **field that has no default value**, new readers wouldn't be able to read data written by old writers, so you would break **backward compatibility**.

Removing a field with / without default value



Removing a new field with / without a default value		
	Backward Compatibility	Forward Compatibility
without		✗
with		✓

If you were to remove a field that has no default value, old readers wouldn't be able to read data written by new writers, so you would break forward compatibility.

Changing the datatype of a field

Changing a datatype of a field is possible, provided that Avro can convert the type.

Changing the field name

Changing a field name	
Backward Compatibility	Forward Compatibility
	

Changing the name of a field is possible but a little tricky: the reader's schema can contain aliases for field names, so it can match an old writer's schema field names against the aliases. This means that **changing field name is backward compatible but not forward compatible**.

This leaves us with the problem of knowing the exact schema with which a given record was written. The best solution depends on the context in which your data is being used:

- In Hadoop you typically have large files containing millions of records, all encoded with the same schema. Object container files handle this case: they just include the schema once at the beginning of the file, and the rest of the file can be decoded with that schema.
- In an RPC context, it's probably too much overhead to send the schema with every request and response. But if your RPC framework uses long-lived

connections, it can negotiate the schema once at the start of the connection, and amortize that overhead over many requests.

- If you're storing records in a database one-by-one, you may end up with different schema versions written at different times, and so you have to annotate each record with its schema version. If storing the schema itself is too much overhead, you can use a hash of the schema, or a sequential schema version number. You then need a schema registry where you can look up the exact schema definition for a given version number.

Example

Let's start with a simple schema example, *user.avsc*:

```
{ "namespace": "example.avro",  
  "type": "record",  
  "name": "User",  
  "fields": [  
    { "name": "userName", "type": "string" },  
    { "name": "favouriteNumber", "type": [ "null", "long" ] },  
    { "name": "interests", "type": { "type": "array", "items": "string" } }  
  ]  
}
```

Note that a schema file can **only contain a single schema** definition. Fully qualified name of the schema is: `<namespace>.<name>` which is `example.avro.User` in this example. Fields `favorite_number` and `favorite_color` are defined as Union data types, and can be null which makes it **optional**.

To write data we use schema defined above to encode the data, below is the python function to write encoded data to a file:

```
import avro.schema  
from avro.datafile import DataFileReader, DataFileWriter  
from avro.io import DatumReader, DatumWriter  
  
schema = avro.schema.parse(open(schema_file, "rb").read())
```

```
def write(schema_file, avro_file):  
    writer = DataFileWriter(open(avro_file, "wb"), DatumWriter(), schema)  
    writer.append({"userName": "Martin", "favouriteNumber": 1337, "interests": [  
    writer.close()
```

This writes to a file and below is the file content:

```
Obj avro.codecnull avro.schema{"type": "record", "name": "User", "namespace":
```

As we can see, encoded data contains schema used in writing the data.

[Avro](#)[Backwards Compatibility](#)[Forward Compatibility](#)[Schema Evolution](#)[Apache Avro](#)[Edit profile](#)

Written by Suresh Podeti

1.2K Followers

Aspiring Software Architect | SDE III (Golang) @JungleeGames| Ex-Byju's | Co-founder @Rupant Tech. | Ex-Synopsys | I.I.T Kharagpur | 7+ Years of Experience

Recommended from Medium